VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Aplikace pro sběr dat ze zařízení připojených přes síť

Applications for Collecting Data from Devices Connected via a Network

2018 Michal Blažek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Blažek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Aplikace pro sběr dat ze zařízení připojených přes síť
Applications for Collecting Data from Devices Connected via a Network

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je naprogramovat aplikaci pro sběr dat ze zařízení připojených přes síť. Aplikace bude poskytovat Web API rozhraní, přes které bude přijímat data z externích zařízení (např. snímač teploty, sledovač počtu vyrobených kusů, apod.).

Serverová část:
1. Služba zajišťující obsluhu klientů:
a) uložení přijatých dat do databáze,
b) poskytnutí dat dle požadavků klientské aplikace.
2. Databáze pro ukládání dat.

Klientská část:
1. Nahlížení do databáze - zobrazení uložených dat.
2. Možnost nastavení filtrů pro zobrazená data.
3. Možnost uložení filtrů.
4. Export dat do externího souboru.

Práce bude obsahovat:
1. Implementaci serverové i klientské části.
2. Programátorskou dokumentaci řešení s využitím diagramů jazuka UML.
3. Uživatelskou dokumentaci aplikace.

Seznam doporučené odborné literatury:

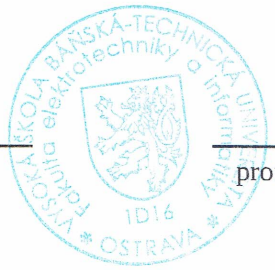
[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí návrhových vzorů. Grada. Praha 2003. ISBN 8024703025
[2] Eeles, Peter, and Peter Cripps. Architektura softwaru. Computer Press, 2011.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Lukáš Kroczek**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018



doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 30. dubna 2018 .....

Rád bych na tomto místě poděkoval vedoucímu mé práce, panu Ing. Lukáši Krozckovi, za nápady a podnětné připomínky, své manželce za velkou podporu, učitelům za předání vědomostí a rodičům a přátelům za dobrá slova.

Abstrakt

Cílem této bakalářské práce "Aplikace pro sběr dat ze zařízení připojených přes síť" je vytvoření programu, který poběží jako serverová aplikace a bude poskytovat službu příjímání dat z různých zařízení, která budou připojena přes síť lokální, či internet. Aplikace bude nabízet správu zařízení, ze kterých bude data přijímat, jejich autorizaci a také správu uživatelů. Práce se zaměří na použité metody a technologie a v závěru zhodnotím funkčnost celého řešení.

Klíčová slova: .NET, sběr dat, zabezpečení, databáze, MVC, WebAPI, testování

Abstract

Goal of this bachelor thesis "Applications for Collecting Data from Devices Connected via a Network" is to create a program that will run on a server and will serve as a data receiver from the devices connected on local network, or over the internet. It will have administration module, which will maintain devices allowed to send the data, device authorization na users administration also. Thesis will focus on technologies and methodology and will sum up the functionality in the end.

Key Words: .NET, data collection, security, database, MVC, WebAPI, testing

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface - Rozhraní pro programování aplikací
MVC	–	Microsoft ASP.NET MVC
M-V-C	–	Model-View-Controller architektura
WebAPI	–	Microsoft ASP.NET WebAPI
Web API	–	Web API architektura
ORM	–	Objektově-relační mapování
EF	–	Entity Framework
CI	–	Continous Integration - Průběžná Integrace
GUI	–	Graphical User Interface - Grafické Uživatelské Rozhraní

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Současnost, v níž žijeme, se vyznačuje rychlostí a efektivitou. Jak stoupá tlak na nižší náklady, a tím pádem vyšší zisky, hlavně výrobních procesů, stále více se rozvíjí automatizace ve výrobě. Aby bylo možné tyto procesy efektivně hodnotit, firmy potřebují mít aktuální a přesná výrobní data. Na druhé straně jsou zde domácí kutilové, kteří si rádi doma sami něco postaví. Například domácí meteo stanici. Z té získávají opět data.

Obě tato odvětví tedy spojuje potřeba získávat informace, ukládat je a dále s nimi pracovat dle potřeby. V této práci se zaměřím na jejich příjem, ukládání a základní zobrazení.

Cílem této práce je vytvořit serverový program, který bude nabízet službu přijímaní dat z různých zařízení připojených přes síť, autentizaci těchto zařízení, ukládání dat a posléze jejich zobrazením uživatelům na vyžádání.

V práci se zaměřím na architekturu celé aplikace tak, aby byla dobře udržovatelná, upravitelná a funkční, dále na nástroje a technologie použité při jejím vývoji a také na problémy, na které jsem narazil, a jejich řešení. V závěru pak zhodnotím postup implementace a jeho vhodnost při vývoji takové aplikace.

2 Architektura

Při vývoji software je možné postupovat vícero možnými způsoby. Základní a pro prvotní vývoj nejrychlejší způsob je mít celou aplikaci napsanou v jednom projektu. V takovém případě se tento projekt stará o celý chod programu, tedy např. o práci s daty (jejich uložení, načtení a zpracování), o business logiku a o prezentaci dat koncovému uživateli. V případě menších aplikací, jako by mohla být např. jednoduchá kalkulačka, je tento přístup ospravedlnitelný, logický a nemusí vyvolat žádné komplikace. V dnešní době jsou však veškeré aplikace čím dál více komplexnější a tím pádem náročnější na údržbu. Znamená to, že se objevují čím dál častěji požadavky například na přenositelnost aplikace mezi platformami. V praxi to znamená, že aplikace musí správně fungovat jak např. v prohlížeči na desktopovém PC, tak také v mobilním zařízení. Zároveň se může objevit potřeba různého uložení dat - ať už lokálně v souborovém systému zařízení, na kterém je spuštěna, nebo do databáze, lokální či vzdálené. V takovém případě se přístup "vše v jednom" ukazuje jako krajně nevhodný z důvodu obtížné údržby a úpravy kódu. V takové situaci je nanejvýš vhodné přemýšlet nad jinou formou formou architektury aplikace. Jako zřejmě nejvhodnější a v současné době nejvíce rozšířená se ukazuje tzv. vrstvená architektura.[1] Jednotlivé komponenty aplikace jsou rozděleny do tzv. horizontálních vrstev. Každá vrstva má na starosti určitou specifickou část aplikace. Cílem této architektury je co nejvíce vzájemně oddělit komponenty, jež jsou každá odpovědná za určitou část funkcionality aplikace. Taková izolace vrstev pak má za následek mnohem jednodušší údržbu aplikace a úpravy, pokud jsou potřeba. Jednotlivé vrstvy aplikace se pak nemusí starat o to, jak jsou další vrstvy ve skutečnosti implementovány, stačí jim vědět, jaké poskytují rozhraní, a proti tomu pak programovat. Ačkoliv není dán pevný počet vrstev, které by měly být použity, většinou jsou tyto vrstvy čtyři. Konkrétní počet se pak bude lišit na základě velikosti a komplexity implementované aplikace. Projekty menšího rozsahu mohou obsahovat vrstvy 3, zatímco větší pět a více. Typicky základní čtyři vrstvy jsou tyto:

- Prezentační vrstva, která má na starosti zobrazení dat uživateli a zároveň příjem požadavků.
- Vrstva business logiky - tato má na starosti veškerou business logiku aplikace.
- Perzistentní vrstva - stará se o základní manipulaci s daty, jejich ukládání a získávání.
- Databázová vrstva - tato vrstva již přímo přistupuje k záznamovému médiu, ať již databázi, či souborovému systému.

Největším přínosem vrstvené architektury je vzájemná nezávislost vrstev. Prezentační vrstva je v hierarchii nejvýše. Má na starosti pouze zobrazování dat uživateli a přijímání jeho vstupu. Pak už neví, jak aplikace s daty manipuluje a jak je ukládá. Business vrstva, která je v hierarchii druhá, pak přijímá data nebo požadavky z prezentační vrstvy a zpracovává je. Pokud je potřeba data uložit, nebo naopak načíst, komunikuje dále s perzistentní vrstvou a nestará se o to, jak

jsou data získávána. Perzistentní vrstva se stará o zpracování dat pro jejich uložení, případně o načtení těch správných. Pro samotnou práci se záznamovým médiem je využita databázová vrstva, jež je v hierarchii nejníže.

V mé aplikaci jsem využil celkem čtyř vrstev, které jsou hierarchicky oddělené. Každá vrstva je implementována v samostatném projektu. Kromě čtyř vrstev jsou v aplikaci další dva projekty popsané dále.

2.1 DBAccess

Tato vrstva je v hierarchii nejníže a slouží k přímému přístupu do databáze. Zde je instalován a konfigurován nástroj Entity Framework (bude popsán později), který zajišťuje mapování mezi entitami v aplikaci a tabulkami v databázi. Tato vrstva tedy zajišťuje práci s daty na nejnižší úrovni.

2.2 Repository

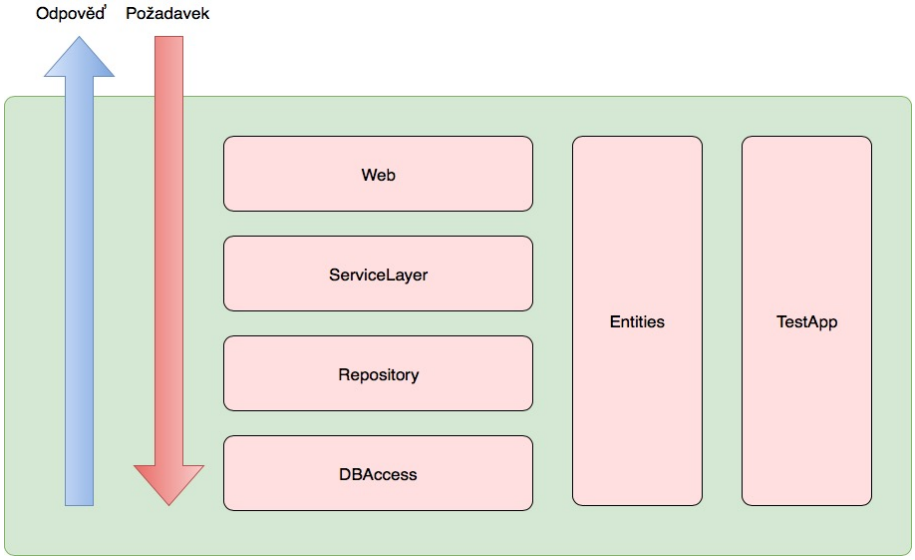
Tato vrstva má na starosti nízkoúrovňové zpracování dat. Posílá požadavky pro načtení dat do databázové vrstvy a zpracovává a upravuje data před jejich finálním posláním databázové vrstvě k uložení. Pro tuto vrstvu jsem použil implementační vzor Repozitář (Repository). [2] Dle něj má každý businessový objekt v aplikaci přiřazenou samostatnou entitu, která se stará o manipulaci s daty, tedy o jejich načtení, uložení, aktualizaci a smazání. Každá taková entita pak iplementuje jeden z principů Extrémního progamování jménem YAGNI, tedy "You Aren't Gonna Need It" (nebudeš to potřebovat). [3] Každou funkci v repositáři, potažmo celé aplikaci, jsem tedy aplikoval až ve chvíli, kdy jsem ji opravdu potřeboval. Neprogramoval jsem něco, o čem jsem si myslel, že by se někdy případně mohlo hodit. Výsledkem tedy je pouze nejmenší nutná množina funkcí, která se v aplikaci používá.

2.3 ServiceLayer

Veškerá logika je uložena ve vrstvě ServiceLayer. Ta má stejnou strukturu, jako vrstva Repository, tedy pro každý businessový objekt existuje právě jedna třída v aplikaci. Tato vrstva definuje hranice aplikace tím, že nastavuje soubor použitelných metod a definuje reakce aplikace na uživatelské požadavky. [4]

2.4 Web

Nejvýše postaveným projektem v hierarchii vrstev je projekt Web. Ten v sobě zahrnuje dvě základní funkcionality. Jednak je to grafické uživatelské rozhraní pro práci s daty a dále poskytuje také WebAPI. V tomto projektu používám framework MVC od společnosti Microsoft. Ten nabízí použití jak webového výstupu pro uživatele, tak WebAPI pro ukládání dat.



Obrázek 1: Architektura aplikace s naznačením směru postupu požadavku skrz jednotlivé vrstvy a odpovědi na něj

2.5 Entities

Tento projekt je jeden ze dvou, které nejsou logicky zařazeny do vrstvové hierarchie projektu, ale je otevřen pro použití v rámci všech projektů v aplikaci. V tomto projektu jsou extrahovány entity, jež vytvořil nástroj Entity Framework dle datové struktury v databázi. Každá entita obsahuje vlastnosti, které odpovídají sloupcům v daných tabulkách v databázi. Ne vždy jsou tyto vlastnosti dostačující, proto projekt obsahuje i další třídy, které jsou potomky vygenerovaných tříd a buď přepisují některé vlastnosti pro potřebu aplikace, nebo naopak přidávají nové.

2.6 TestApp

Tento projekt je poslední z celé aplikace a také stojí mimo hierarchii vrstev v aplikaci. Tento projekt je určen pro implementaci jednotkových testů. Pro jednoduchou a přímočarou implementaci jsou instalovány nástroje NUnit, FakeItEasy a FluentAssertions. Použité testy a nástroje budou popsány dále.

2.7 Business objekty aplikace a jejich interakce

Základními objekty aplikace jsou: zařízení (Device), prvky zařízení (DeviceColumns), API klíč (ApiKey), data (DataRow), uživatelé (DCUser), role (Role) a oprávnění (DataEntitlement). Data jsou ukládána v objektu DataRow, který je propojen s objektem Device. Při vkládáni dat (DataRow) je autentizace prováděna vzhledem k API klíči přiřazenému zařízení. Uživatel, jenž musí být do aplikace přihlášen, aby mohl provádět jakoukoliv práci, si může zobrazit data pro ta zařízení, která má přiřazena díky oprávnění (DataEntitlement).

2.8 Zpracování výjimek

V každém programu mohu nastat, a také nastanou, výjimečné stavy, které mohou být způsobeny aktivitou uživatele, nebo dalšími vlivy, nad kterými uživatel nemá moc. V případě takových stavů vyvolaných uživatelem se může jednat například o špatně zadaná vstupní data, nebo kombinaci vzájemně nekompatibilních akcí. V případě situací, na které uživatel vliv nemá, se může jednat o ukončené síťové připojení nebo o chybu zápisu dat na disk.

Ať nastane jakákoliv nestandardní situace, program by měl být vývojářem napsán tak, aby tyto chyby nezpůsobily pád aplikace a ta se s problémem vyrovnala a uživatel mohl dále pracovat. Případně, aby byl patřičně informován o problému. V takové chvíli není žádoucí, aby se zobrazila chyba, kterou vygeneruje systém, protože ta může být plná technických detailů, jež budou pro běžného uživatele nesrozumitelné a hlavně nepodstatné. Platforma .NET, kterou jsem pro implementaci bakalářské práce vybral, poskytuje velmi dobrý nástroj, jak se s chybovými stavy vyrovnat. Tento nástroj je postaven na mechanizmu zvaném obsluha výjimek. [5]

Výjimka je objekt, jenž je vytvořen v případě výskytu chybového stavu. Obsahuje v sobě informace, které vývojáři pomohou dojít k původu chyby a patřičně se s ním vyrovnat, aby aplikace dále fungovala. Ačkoliv je možné definovat své vlastní výjimky, platforma .NET obsahuje širokou škálu předdefinovaných výjimek, které je možné použít. V mnou vytvořené aplikaci jsem nemusel žádné vlastní výjimky vytvářet, ale bylo možné použít některé z předdefinovaných a pouze jim předat vlastní text chyby.

Obsluha výjimek probíhá takovým způsobem, že je zachycena a zpracována buď přímo v místě, kde chyba nastala, nebo výše v hierarchii volajícího kódu. Můžeme tedy tímto způsobem oddělit kód, kde chyba nastala, od kódu, který ji zpracovává.

Další možností, jak chyby zpracovávat, je použití návratových hodnot. To funguje tak, že volaná metoda podle stavu, jak dopadl kód v jejím těle, vrací volající metodě příslušnou hodnotu. Tento přístup je v pořádku ve chvíli, kdy se návratové hodnoty dále zpracovávají ve volající metodě. Avšak pokud se takto zároveň vrací hodnoty pro výjimečné stavy, vzniká takto těsná vazba samotného kódu metody, tedy toho, co je její primární účel, a řešení výjimečných stavů. V případě užití návratových hodnot v aplikaci s vrstvovou architekturou se situace dále komplikuje, protože chybový stav musí být zpracován a případně předán dál v každé vrstvě. Zdrojový kód tak s každou vrstvou získává na objemu. Tato potřeba odpadá v případě užití výjimek, jelikož ty "probublávají"mezi vrstvami až k původní volající metodě. Odpadá tak potřeba kódu, jenž jen předává chybu výše v hierarchii vrstev.

Použití výjimek tedy bylo logickým krokem ve chvíli, kdy jsem řešil, jak validovat data zadávaná uživateli a informovat je o výsledku. Samotná validace neprobíhá v prezentační vrstvě, ale ve vrstvě business logiky. V případě, že validace není úspěšná, vytvářím v daném místě explicitně objekt výjimky, kterému nastavuji příslušnou chybovou zprávu srozumitelnou uživateli. V prezentační vrstvě je pak výjimka zachycena, původní požadavek zadaný uživatelem je zrušen a je mu zobrazena chybová zpráva.

2.9 Vkládání závislostí

Jedním z cílů výběru vrstvové architektury bylo, aby v budoucnu, v případě potřeby, bylo možné relativně jednoduše vyměnit jednu z vrstev programu za jinou, bez potřeby refaktorizace velké části kódu. Do značné míry tento požadavek usnadňuje použití rozhraní. I přes to vzniká vazba mezi jednotlivými vrstvami tím, že při vytváření třídy musíme této třídě předat konkrétní implementaci jiné třídy, na které je závislá. Pokud by tedy bylo potřeba vyměnit konkrétní implementační třídu, musela by se v celém kódu vyhledat všechna místa, kde se tato třída inicializuje a upravit tyto inicializace. Tento přístup je nejen časově náročný, ale také velmi náchylný k chybám v případě opomenutí některých změn.

Lepší a v moderním programování hojně využívanou možností, jak tento problém vyřešit, je použití návrhového vzoru Inversion Of Control. Ten funguje na tom principu, že objekt jedné implementující třídy není vytvořen pevně v rámci druhé třídy, ale je této druhé třídě předán v parametru konstruktoru. Tímto způsobem ztrácí kontrolu nad výběrem implementující třídy třída s daným rozhraním, ale předává se výše v hierarchii volajícího kódu. Můžeme tak velmi snadno a rychle měnit chování našeho programu. Výše popsaný princip najde své uplatnění nejen při změně programu, ale velmi zjednodušuje a urychluje testování programu.

Mějme například třídu v perzistentní vrstvě, jež zpracovává určitým způsobem data, která ji jsou předána, a následně tato data uloží do databáze. Chceme-li tuto třídu otestovat, zajímá nás, zda funguje správně zpracování dat, avšak již ne zápis do databáze. Máme k tomu dva důvody. Za prvé nechceme, aby se nám testovací data zapisovala do databáze, a za druhé k otestování samotného zápisu do databáze má mít své testy vrstva, která má tuto činnost na starosti. Pokud máme třídu napsanou dle návrhového vzoru Inversion of Control, jednoduše v testu předáme jako parametr třídy starající se o zápis do databáze, nějaký jiný objekt, který bude mít stejné rozhraní, ale nebude do databáze skutečně zapisovat.

Návrhový vzor Inversion of Control je velice obecný. Ve své práci jsem se zaměřil na konkrétnější způsob, který se nazývá Dependency Injection, tedy Vkládání závislostí. V tomto případě programátor neposílá objekty implementujících tříd do konstruktorů jiných tříd sám, ale zajišťuje to pro něj samostatný framework. Ten musí být nakonfigurován, ať již pomocí zdrojového kódu, nebo např. pomocí XML souborů, tak, aby věděl, která třída v projektu implementuje které rozhraní.

V mé práci jsem si jako Dependency Injection framework vybral Castle Windsor. Tento framework, vytvořený skupinou programátorů v rámci projektu Castle, je jedním z mnoha Dependency Injection frameworků, které je možné pro platformu .NET získat. Tento framework jsem vybral z důvodu kombinace faktorů, mezi které patří jeho oblíbenost, dobrá dokumentace a jednoduchost použití. [6]

Konfiguraci tohoto frameworku jsem prováděl v kódu. Bylo třeba pouze nastavit, která třída implementuje které rozhraní. Celé nastavení bylo dílem několika řádků a fungovalo bez problému. Ukázka jednoduchosti tohoto nastavení následuje:

```
public void Install(IWindsorContainer container, IConfigurationStore store)
        {
            container.Register(Component.For<IDeviceService>().
                LifestyleTransient().ImplementedBy<DeviceService>());
            container.Register(Component.For<IDataRowService>().
                LifestyleTransient().ImplementedBy<DataRowService>());
        }
```

Výpis 1: Ukázka registrace rozhraní a implementující třídy

Pokud pak Castle Windsor sám vytváří instance pro příslušná rozhraní, sám se také stará o jejich likvidaci. Pokud ale máme tento framework nakonfigurován tak, že chceme pro každý požadavek vytvořit novou instanci a ne použít již existující, a zároveň chceme explicitně získat instanci pro nějaké rozhraní, musíme ji po použití také sami zrušit. Pokud bychom tak nečinili, instance by se nám hromadily v paměti, která by byla postupně zaplňována. Narazili bychom na problém zvaný Memory leakage.

Následuje ukázka kódu pro ověřování uživatele, v jejímž rámci získávám instanci pro dané rozhraní a po použití tento zdroj uvolním.

```
public override bool ValidateUser(string username, string password)
        {
            IDCUserService userService = IocContainer.Container.Resolve<
                IDCUserService>();
            DCUser user = userService.GetByLoginAndPassword(username, password);

            IocContainer.Container.Release(userService);

            return user != null;
        }
```

Výpis 2: Získáni instance pro rozhraní a její uvolnění

2.9.1 Problémy při implementaci

Castle Windsor musí být při implementaci správně nakonfigurován. Jeden z problémů, na které jsem narazil, byla nefunkčnost aplikace ve chvíli, kdy se sešly dva požadavky na zápis či čtení v rámci databáze ve stejnou chvíli. Castle Windsor má jaké své originální nastavení takové, že pokud je požádán o instanci objektu, který má ve své režii, vrátí vždy jednu a tu samou instanci, tedy jedináčka (singleton). Pokud pak nastane situace, že se setkají dva požadavky na zápis do databáze, Castle Windsor vrátí vždy stejnou instanci databázového kontextu. A takový zápis selže a vyhodí výjimku. Bylo tedy potřeba nastavit správný takzvaný lifecycle, tedy životní

cyklus, což je způsob, jakým Castle Windsor zachází s instancemi. Má-li pro každý požadavek vytvořit novou instanci nebo používat stále jednu, případně má li být instance vytvořena pro každé vlákno zvlášť, apod.

Jako správné nastavení se ukázalo nastavení Transient, které pro každý požadavek vytváří novou instanci, nikde nepoužívá jednu a tu samou dvakrát. Toto nastavení také bylo potřeba aplikovat na všechny vrstvy nad databázovým kontextem, nestačilo to tedy pouze pro zmíněný databázový kontext. Správné nastavení lze vidět ve výpisu kódu 1.

3 Rozhraní

Aplikace poskytuje pro komunikaci s okolím dvě rozhraní. Pro uživatelskou práci, nahlížení na data, administraci jsou to webové stránky, postavené na architektuře ASP.NET MVC. Rozhraní pro venkovní zařízení, přes která mohou do aplikace posílat data, je řešeno pomocí ASP.NET WebAPI. Obě rozhraní jsou zavedena v prezentační vrstvě.

3.1 MVC

M-V-C, nebo-li Model-View-Controller, je architektonický návrhový vzor. Tento přístup rozděluje aplikaci na tři základní komponenty, přičemž změna kterékoliv z nich by měla v co nejmenší míře ovlivnit zbylé dvě. Každá komponenta má na starosti samostatnou část aplikace.

3.1.1 Model

Model je v rámci M-V-C datová struktura, která by měla odpovídat prvkům reálného světa, nebo konkrétněji doménového modelu aplikace. V rámci této vrstvy by měla být uložena také veškerá business logika a validace.

Entity v této vrstvě mají vazby jen mezi sebou, nedrží si žádné reference na vrstvy Controller, nebo View.

Model v mém řešení je prakticky složen ze dvou částí. První je vrstva Entities, která pouze reprezentuje objekty doménového modelu. Business logika je pak uložena ve vrstvě ServiceLayer.

3.1.2 View

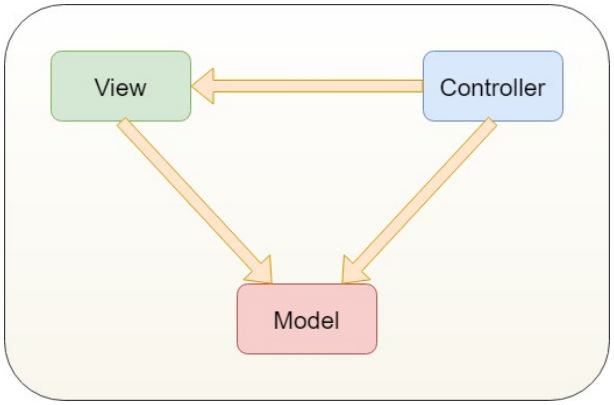
View prezentuje vrstvu, která se stará o grafický reprezentaci modelu pro uživatele. V této vrstvě by neměla být žádná aplikační logika. View může obsahovat referenci na entity z vrstvy Model, aby mohlo data správně zobrazovat. Naopak referenci na Controller neobsahuje, z něj pouze přijímá data.

3.1.3 Controller

Controller reprezentuje vrstvu mezi View a Model a poskytuje metody dostupné pro uživatele. Pokud uživatel provede ve View nějakou akci, řízení programu je předáno Controlleru, který požadavek zpracuje a potřeby přepošle dále do Modelu. Jakmile jsou data v Modelu zpracována, tok programu se vrátí zpět do Controlleru. Ten poté předá řízení společně s potřebnými daty do View, které následně vygeneruje HTML výstup, který je zobrazen uživateli.

3.2 WebAPI

WebAPI je framework, s jehož pomocí lze programovat architekturu Web API, tedy API dané aplikace přístupné přes protokol HTTP. WebAPI funguje v rámci MVC frameworku jako jeho



Obrázek 2: Zobrazení vrstev modelu MVC a jejich vzájemných referencí

přirozená součást. Využívá stejnou strukturu vrstev Controller a Model. Rozdíl mezi MVC a WebAPI Controllerem je v návratové hodnotě, kterou Controller vytvaří. Zatímco v MVC je jako výstup použito View, které pak generuje HTML výstup pro uživatele, v případě WebAPI View chybí a výsledkem metody je HttpMessageResponse, respektive data.[7] Tím je splněn základní předpoklad pro REST architekturu. WebAPI je tedy vhodné pro tvorbu RESTful aplikací.

REST architektura je způsobem komunikace mezi programy přes HTTP protokol, která je často používána při vývoji webových služeb. REST nevyžaduje žádné pevné pravidlo, jak by měla být konkrétně implementována služba, ale spíš dává obecnější požadavky a samotnou implementaci nechává na vývojáři.[8]

3.2.1 Formát požadavku

Aby má aplikace mohla data přijmout, musí mít webový požadavek pevně daný formát. V řetězci obsahujícím požadavek, musí být krom správné adresy, na které aplikace běží, a portu také následující parametry:

- /Values/Insert/

  Tímto je zajištěno, že požadavek bude zpracován správným Controllerem.

- apiKey

  API klíč přiřazený zařízení

- deviceName

  Název zařízení, ze kterého přichází data

- date

  Datum, které má být uloženo společně s daty. Tento parametr je volitelný. Nebude-li součástí požadavku, uloží se datum přijetí dat do aplikace. Formát musí být ve tvaru: mm/dd/yyyy

- values

  Samotná data, která mají být uložena. Tento parametr se musí opakovat tolikrát, kolik je v databázi uloženo pro zařízení sloupců.

Požadavek, který by mohl být přijat, pokud by byl následně úspěšně validován, by mohl vypadat například takto:

```
http://localhost:55405/Values/Insert/?apiKey=f06c0109-dcc8-4c3a-a94a-596eb723a975
                    &deviceName=Zarizeni1&values=23&values=56
```

V příkladu je tedy zařízení s názvem "Zarizeni1", API klíčem "f06c0109-dcc8-4c3a-a94a-596eb723a975", datum není vyplněno, použije se tedy aktuální den, a pro zařízení jsou v databázi uloženy dva sloupce, do kterých budou uloženy hodnoty "23" a "56".

3.2.2 Návratové kódy

Jak jsem již zmínil, WebAPI, na rozdíl od MVC, nevrací View, ale status kódy, jak zaslaný požadavek dopadl. Status kódy použité v mé aplikaci jsou:

- 200 OK

  Data byla úspěšně uložena.

- 400 Bad request

  Vyskytla se chyba a data nebyla zpracována. U status kódu je přidán také text, který vysvětluje, proč nebyl požadavek přijat. V rámci této odpovědi mohou původnímu odesílateli dojít tyto zprávy:

  - Chybí některé povinné parametry (obrázek 8)
  - API klíč neexistuje
  - API klíč není platný pro aktuální datum
  - API klíč a zařízeni se neshodují
  - Neznáme zařízení
  - Počet hodnot pro data se neshoduje s počtem sloupců u zařízení (obrázek 9)

4 Databáze a přístup k datům

Prakticky u každé aplikace je potřeba nějakým způsobem ukládat data. V závislosti na různých faktorech se pak vývojář může rozhodnout, v jakém formátu bude data ukládat. Pokud je potřeba uložit pouze nastavení aplikace a nebude třeba je zabezpečit proti přečtení, bude stačit vytvořit lokální soubor a do něj nastavení uložit. Pokud však bude potřeba ukládat větší množství dat a pracovat s nimi, tento způsob zřejmě nebude ten nejlepší.

Při výběru úložiště jsem se tedy rozhodoval mezi těmito základní způsoby:

- Textové soubory

  Textové soubory (čistý text, XML, JSON) jsou oblíbené pro ukládání nastavení aplikace, protože jsou pak velmi jednoduše spravovatelná. Nevýhodou takového uložení je zabezpečení dat (žádné), ale také rychlost a komfort práce, např. vyhledávání dat. Takový formát je krajně nevhodný pro ukládání velkých množství dat, se kterými se bude často pracovat.

- Binární soubory

  Oproti textovým souborům mají výhodu menší náročnosti na paměť a jsou lépe zabezpečené, protože je potřeba znát formát, v jakém jsou uložené a otevřít je správným programem. Nevýhody jsou však podobné, jako u textových souborů - tedy rychlost, náročnost na vyhledávání. Tento formát je tedy opět velmi nevhodný k ukládání většího množství dat, jaká bude produkovat má aplikace.

- Databáze

  Databáze poskytují jak možnost ukládat velké množství dat, tak i rychlou a komfortní práci s nimi. Pro svou práci jsem tedy vybral implementaci ukládání dat do relační databáze.

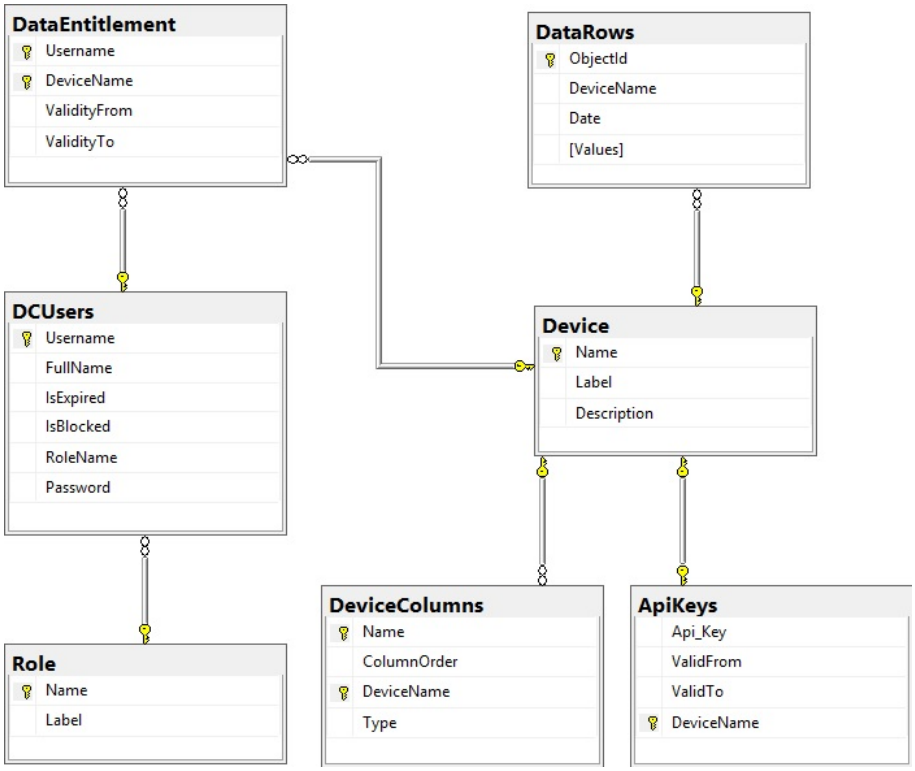
Po rozhodnutí, jaký způsob uložení zvolit, jsem se musel rozhodnout, jakou databázi použít. Mezi třemi všeobecně nejoblíbenějšími jsou databáze Oracle, MySQL a Microsoft SQL Server. [9]. Vzhledem k implementaci celé aplikace na platformě .NET padla volba logicky na Microsoft SQL Server.

4.1 Struktura databáze

Architektura aplikace následuje návrhový vzor Doménový model [4] a tomuto uspořádání odpovídá i struktura databáze. Každá entita v rámci architektonické vrstvy Entities má v databázi právě jeden protějšek, který se shoduje jak názvem, tak atributy, tedy v případě databáze sloupci.

4.2 Objektově-relační mapování

Přístup do databáze je implementován ve vrstvě DBAccess. Používám návrhový vzor přístupu k datům ORM, tedy objektově-relační mapování. To funguje na takovém principu, že mapuje,



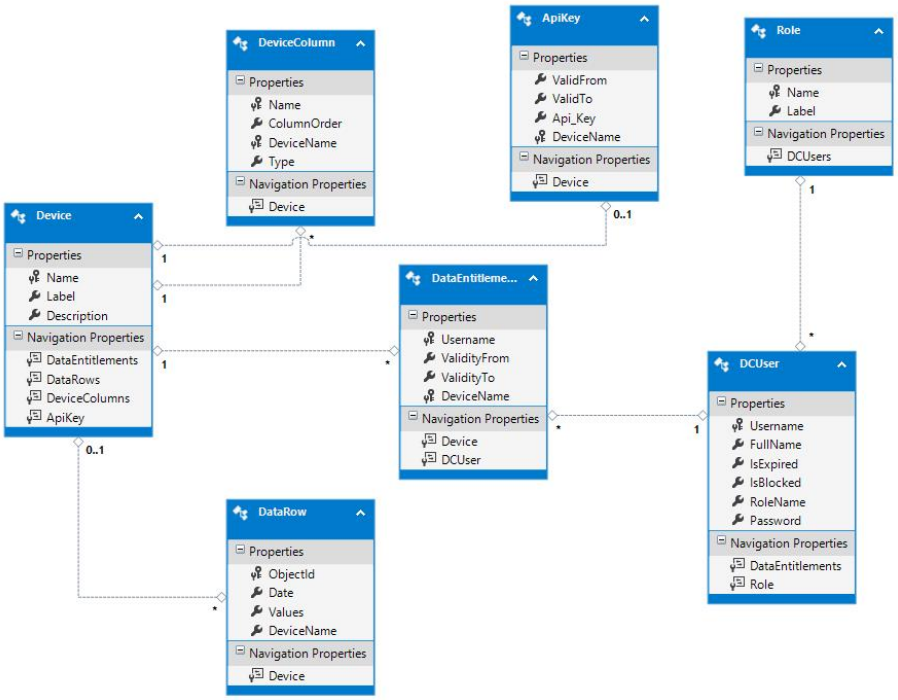
Obrázek 3: Diagram databáze se vztahy mezi jednotlivými tabulkami

tedy propojuje, entity v programu s tabulkami v databázi. Základní možnost, jak docílit fungování ORM, je naprogramovat celé mapování mezi tabulkami a entitami vlastními silami. Druhou možností je použití některého z existujících frameworků, které mapování značně ulehčí. Výhodou samostatné implementace bude plná kontrola nad procesem, nevýhodou však bude časová náročnost a fakt, že výhody samostatné implementace nepředčí rychlost, komfort a ověřené fungování existujících frameworků.

K dispozici je opět celá řada nástrojů, která objektově-relační mapování usnadňuje. Při rozhodování, který z nástrojů použiji, jsem se opět držel nástrojů společnosti Microsoft a zvolil Entity Framework. Tento framework má svá určitá omezení, například prakticky žádnou kontrolu nad generovanými SQL dotazy nebo pomalejší první spuštění, případně pevné užití návrhového vzoru Unit of Work [4] v rámci Entity Frameworku. V mé aplikaci by však nic z těchto potencionálních komplikací neměl být problém a testování aplikace v provozu ukázalo, že tomu tak opravdu je.

4.3 Entity Framework

První verzi Entity Frameworku, číslo 3.5, vypustila společnost Microsoft do světa v roce 2008 v rámci frameworku .NET 3.5 SP1 a Visaual Studia 2008 SP1. [10] V první verzi poskytoval EF pouze základní podporu pro ORM a princip, na kterém fungoval byl Database-First. To znamená, že bylo potřeba nejdříve vytvořit databázi a na jejím základě vytvořil EF v programu



Obrázek 4: Diagram entit se vzájemnými vztahy mezi nimi

entity. V poslední stabilní verzi 6.2.0 (stav ke dni 26.4.2018) je již EF daleko rozvinutějším nástrojem.

Mezi zajímavé vlastnosti řadím možnost zprovoznit EF metodou Code-First. V takovém případě nejdříve programátor vytvoří v programu entity dle doménového modelu, s kterým pracuje, a následně již EF sám vytvoří databázové tabulky. Vývojář se tak prakticky může zcela soustředit na kód samotný a musí již pouze nastavit connection string, tedy připojovací řetězec, aby EF věděl, jak se připojit do databáze.

Ve svém projektu jsem ale použil metodou Database-First, jelikož jsem chtěl mít nad tabulkami vytvořenými v databázi plnou kontrolu. Tvorba entity pak proběhla automaticky dle očekávání. Ve Visual Studiu je pak dobře vidět náhled entit a jejich vazeb, podobně jako u tabulek v SQL Server management studiu na obrázku 4.

4.3.1 Problémy při implementaci

Problém při implementaci DBAccess vrstvy nastal ve chvíli, kdy jsem potřeboval v ostatních vrstvách referencovat entity vytvořené Entity Frameworkem. Tím by vznikla taková vazba mezi vrstvami, jaké jsem se chtěl vyhnout. Řešením nakonec bylo tyto entity vytáhnout do samostatného projektu, tedy vrstvy, a v konfiguraci EF nastavit cestu k těmto entitám. [11]

Další problém nastal při aktualizaci entit Entity Frameworkem. Pro to, abych měl správně funkční validaci dat a aby se správně zobrazovaly názvy pro jednotlivé položky entit, musel jsem nastavit v těchto entitách některé atributy. Příklad nastavení atributů pro pole reprezentující

datum je ve výpisu číslo 3. Avšak ve chvíli, kdy jsem potřeboval aktualizovat objektově-relační mapování podle databáze a přegenerovat třídy, Entity Framework veškeré mé změny přepsal, tedy odstranil mnou nastavené atributy. V této chvíli se ukázalo být velice efektivní, že jsem používal systém správy verzí (Bitbucket), s jehož pomocí bylo zpětné vrácení mých změn otázkou několika okamžiků.

```
[DataType(DataType.Date)]
[DisplayFormat(ApplyFormatInEditMode = true, DataFormatString = "{0:yyyy-MM-dd}
    ")]
[Display(Name = "AKValidFrom", ResourceType = typeof(Resources.DisplayNames))]
public Nullable<System.DateTime> ValidFrom { get; set; }
```

Výpis 3: Nastavení atributů pro pole typu Date

5 Zabezpečení aplikace

V posledních letech je stále více a více kladen důraz na zabezpečení aplikací. I u mé aplikace, přes to, že by ze své povahy mohla běžet pouze na lokální síti v rámci domácnosti nebo firmy, bylo zapotřebí toto téma vyřešit. Zabezpečení jsem vyřešil na několika úrovních, popsaných níže.

5.1 Uživatelé, role a oprávnění

Zabezpečení aplikace na úrovni uživatelského přístupu jsem vyřešil použitím mechanismu uživatelských účtů. Ke každému účtu se váže role a datové oprávnění.

5.1.1 Uživatelské účty

Jako první krok zabezpečení jsem implementoval správu uživatelů. Ta je velmi dobře podporována v rámci frameworku MVC. Ke zprovoznění je potřeba provést několik kroků.

Prvním z nich je implementace dvou tříd. Jedna, poskytující službu uživatelských účtů, musí implementovat rozhraní `MembershipProvider`. Toto rozhraní poskytuje metody a atributy pro to, aby správa uživatelů fungovala správně. Ukázka implementace je ve výpisu 2. Druhým rozhraním, které je třeba implementovat, je rozhraní `RoleProvider`, které je podobné, jako `MembershipProvider`, ale má na starosti role v systému, místo uživatelských účtů.

Dalším krokem je konfigurace pro aplikaci, ve které je třeba správu uživatelských účtů aktivovat a říct systému, které třídy implementují potřebná rozhraní `RoleProvider` a `MembershipProvider` a také, jaký způsob autentizace bude použit. V případě mé aplikace je to mód Forms, který předpokládá existenci stránky, ve které budou uživatele zadávat své uživatelské jméno a heslo, aby se do aplikace přihlásili.

```
<authentication mode="Forms">
      <forms loginUrl="/Login" requireSSL="true" />
    </authentication>
    <membership defaultProvider="DCMembershipProvider">
      <providers>
        <clear />
        <add name="DCMembershipProvider" type="DataCollector.Authentication.
            DCMembershipProvider"/>
      </providers>
    </membership>
    <roleManager enabled="true" defaultProvider="DCRoleProvider">
      <providers>
        <clear />
        <add name="DCRoleProvider" type="DataCollector.Authentication.
            DCRoleProvider"/>
```

```
    </providers>
  </roleManager>
```

Výpis 4: Konfigurace správy uživatelských účtů v souboru `web.config`

5.1.2 Role

Aby správa uživatelských účtů fungovala správně, je třeba mít v aplikaci definované také role, které mohou uživatelé mít.

Role definuje, jaké akce má uživatel právo v aplikaci provádět. V rámci implementace Controlleru je pak třeba uvést, zda má být daný Controller (případně samostatná metoda) dostupná jen přihlášeným uživatelům. Doplňující informací je možno dále omezit konkrétní role, které budou mít k dané akci přístup.

V systému jsou zavedeny dvě role, které mohou být uživateli přiděleny v rámci správy uživatelského účtu:

- Admin

  Uživatel s touto rolí vystupuje v aplikaci jako administrátor. Může tedy spravovat uživatelské účty, zařízení a také vidí uložená data všech zařízení.

- User

  Běžný uživatel, který po přihlášení může pouze vidět data, na která má oprávnění.

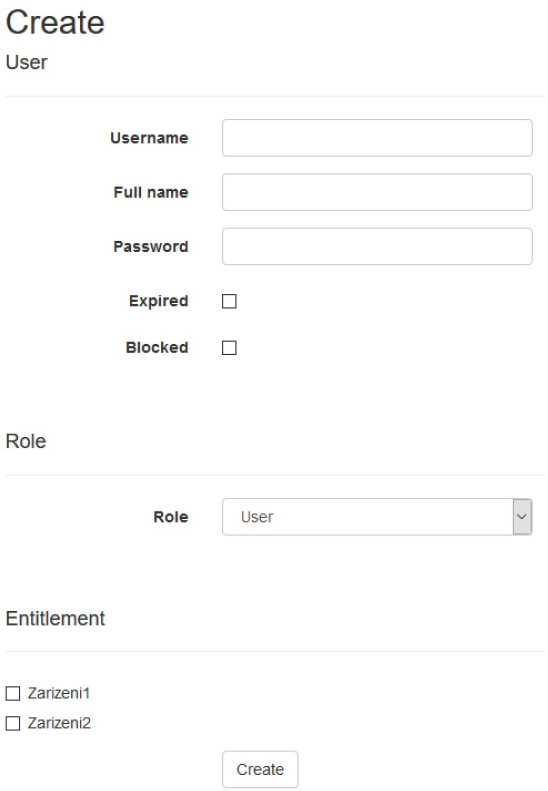
5.1.3 Oprávnění

Dalším prvkem zabezpečení, jenž jsem implementoval, bylo datové oprávnění, tedy kteří uživatelé mohou vidět data kterých zařízení.

Motivací pro toto zabezpečení je, že není žádoucí, aby všichni uživatelé viděli data všech zařízení v systému. Implementace je taková, že podobně, jako má uživatel přiřazenou roli, má také přiřazené zařízení, jejichž data si může zobrazit. Toto přiřazení probíhá v rámci správy uživatelského účtu a může je nastavit pouze administrátor. Náhled vytváření uživatele včetně nastavení role a oprávnění je na obrázku 5.

5.2 HTTPS/SSL

Bezpečnost je ve světě IT velké téma, jelikož data, ať již o uživatelích, nebo průmyslová, základní potřeby, s kterými pracují. V této sekci dále popíšu základní způsob zabezpečení dat při jejich přenosu sítí a které části mé aplikace zabezpečeny jsou, které ne a proč ne.



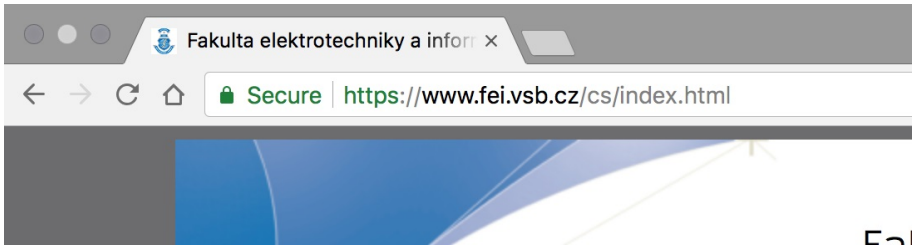
Obrázek 5: Vytvoření uživatele

5.2.1 SSL

SSL (Secure Socket Layer) je základní technologie umožňující zabezpečení dat při jejich přenosu mezi různými systémy. [15] Jako systémy jsou v tomto případě označeny například klient (webový prohlížeč) a server, nebo dva různé servery. Jedná se o protokol, který pracuje mezi transportní vrstvou (TCP/IP) a vrstvou aplikační (HTTP) a data před samotným přenosem šifruje, takže nejsou čitelná třetí stranou, tedy útočníkem, který by se je mohl pokusit zachytit. Lze tak tedy bezpečně přenášet citlivá data jako jsou osobní informace o uživatelích, čísla kreditních karet, apod.

5.2.2 TLS

TLS (Transport Layer Security) je novější, více zabezpečená verze SSL. TLS ve verzi 1.0 je pouze málo odlišná od SSL verze 3.0. Jedním ze základních rozdílů je možnost vytvářet virtuální webové servery, což SSL neumožňuje. [16] TLS tak řeší problém, aby na jedné IP adrese mohlo běžet více domén bez nutnosti pro každou z nich vytvářet samostatnou veřejnou IP adresu. Technologie, která tomu napomáhá, se jmenuje SNI (Server Name Indication) a funguje tak, že místo ověřování IP adresy ověřuje jméno hostitele. [17]



Obrázek 6: Informace o zabezpečení webové stránky zobrazeá v prohlížeči

5.2.3 HTTPS

HTTPS je protokol zabezpečená verze protokolu HTTP. Písmeno S na konci zkratky znamená Secure, tedy bezpečný. HTTP protokol se stará o přenos dat mezi webovým prohlížečem a webovým serverem.

Je-li certifikát SSL či TLS na serveru správně nainstalován, zobrazí se tato informace uživateli (obrázek 6). V případě některých prohlížečů se zobrazí zámek a informace, že připojení je zabezpečeno. Uživatel si pak může dále zobrazit veškeré dostupné informace o certifikátu (obrázek 7).

5.2.4 Zabezpečení přenosu dat v mé aplikaci

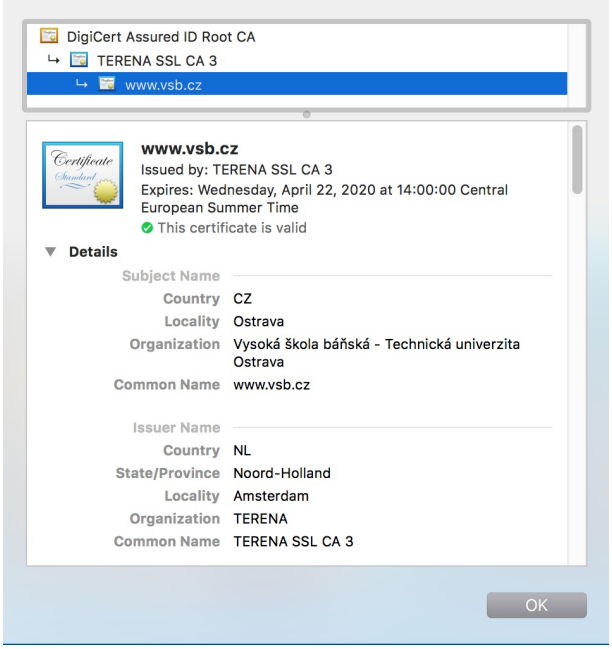
V mé práci jsem využil obě varianty přenosu dat, tedy jak šifrovanou (HTTPS), tak nešifrovanou (HTTP). Celá část aplikace, která je dostupná pro uživatele přes webový prohlížeč, tedy zobrazení dat, administrace, funguje na protokolu HTTPS. WebAPI, které přijímá data ze zařízení, používá nešifrovaný přenos, tedy HTTP. To je z důvodu větší kompatibility zařízení, jelikož některá starší zařízení nemusí šifrovaný přenos podporovat.

5.3 API klíče

Aby mohla zařízení posílat data do aplikace přes rozhraní WebAPI, musí probíhat kontrola, zda jsou tato zařízení uložena v systému, zda jsou opravdu ta zařízení, za která se vydávají a zda mají oprávnění data do systému ukládat. Všechny tyto kontroly proběhnout ve chvíli, kdy data dorazí přes HTTP požadavek do aplikace.

Nejdříve se systém pokusí nalézt v databázi zařízení, které je v požadavku s daty uvedeno. Požadavek obsahuje název zařízení. V databázi je název primárním klíčem, tato hodnota je tedy dostačující. Je-li nalezena shoda, pokračuje se s další validací.

V dalším kroku je vyhledán API klíč uvedený v požadavku. Je-li takový záznam v databázi nalezen, je tento klíč načten. Další kontrolou je porovnání, zda dvojce zařízení-klíč odpovídá záznamům spárovaným v databázi. Finální krok je pak validace platnosti API klíče v daný den poslání dat. Pokud všechny tři kontroly projdou úspěšně, data mohou být vložena do systému.



Obrázek 7: Zobrazení informací o SSL certifikátu pro www.vsb.cz

Generování API klíče probíhá při vytváření či úpravě zařízení. Pole pro tvorbu klíče je needitovatelné, vyplní se na základě stisku tlačítka, které pomocí javascriptové funkce API klíč vygeneruje. Uživatel také může nastavit platnost API klíče.

5.3.1 Formát API klíče

API klíč je reprezentován UUID, tedy Universally Unique Indentifier, známého také jako GUID. Jedná se o 128 bitovou sekvenci znaků, která by měla zaručovat unikátnost záznamu. Tato sekvence je kódována jako 32 hexadecimálních znaků ve formátu "`xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx`". V aplikaci generuji UUID verze 4 [12], formát tedy je "`xxxxxxxx-xxxx-4xxx-xxxxxxxxxxxx`". Verze 4 znamená, že hodnoty v sekvenci jsou generovány náhodně nebo pseudonáhodně.

5.3.2 Platnost API klíče

Při tvorbě či úpravě zařízení lze nejen vygenerovat samotný API klíč, ale lze mu také nastavit časovou validitu. Při autentizaci zařízení se pak kontroluje, zda je datum poslání dat větší, či rovno datu "od" nastaveného u API klíče a zároveň menší nebo rovno datu "do" nastaveného u API klíče.

5.4 Zabezpečení hesel

Uživatelské účty jsou, stejně jako ostatní data, uloženy v databázi. Součástí účtu je také heslo. To může být v databázi uloženo několika různými způsoby. [13] Ty rozepíšu dále, stejně jako konečný způsob, jak hesla ukládám.

5.4.1 Čistý text

Nejjednodušší způsob, jak hesla ukládat, je jako čistý text. Tento způsob je nejjednodušší, nejrychlejší, ovšem také nejméně bezpečný. Veškeré zabezpečení pak stojí na prostředí kolem hesla - tedy na zabezpečení databáze samotné. Pokud se kdokoliv dostane do databáze, získá tak okamžitě hesla ke všem účtům, jež jsou v ní uložena.

5.4.2 Šifrovaná hesla

Způsobem, jak přidat heslům vyšší stupeň bezpečnosti je jejich šifrování. Tato metoda funguje tak, že ještě v prohlížeči se heslo zašifruje dle nějakého klíče. Na server je pak odesláno již v zašifrované formě. Server pak dle potřeby může heslo dešifrovat dle klíče. Problém u takového způsobu je ten, že klíč k dešifrování bývá často uložen na stejném serveru, jako hesla. Pokud se tedy opět někdo dostane do takové databáze, nemusí již poté vynaložit příliš velké úsilí k tomu, aby získal i hesla.

5.4.3 Hashovaná hesla

Hashování funguje na podobném principu, jako šifrování. To znamená, že zadané heslo změní na zakódovanou sekvenci znaků. Rozdíl proti šifrování je v tom, že při hashování existuje pouze jedna cesta, tedy získat hash z hesla. Z výsledného hashového řetězce není možné získat heslo zpět. Teoreticky se tedy jedná o velmi bezpečný způsob, jak uchovávat hesla. Způsob, jakým se útočníci snaží prolomit heslo, se nazývá Duhová tabulka (Rainbow table). [13] Duhové tabulky jsou v podstatě obrovské databáze (i terabajty dat), ve kterých jsou uložena hesla v čistém textu a k nim pasující hash řetězce. Pokud se tedy útočník dostane k databázi, ve které jsou uložená zaheshovaná hesla, pokusí se je vyhledat v Duhových tabulkách. Je dokonce možné, že dvě různá hesla budou mít stejný hash řetězec, což však útočníkovi nevadí, protože mu stačí pouze výstup.[14]

Největší slabinou takových hesel tedy není jejich komplexita, ale délka. V praxi to znamená, že heslo “He$1o1“ bude útočníkem prolomeno mnohem rychleji, než “auto barva slunce pravda“.

5.4.4 Hashovaná hesla se solí

Způsobem, jak zvýšit bezpečnost hesel, je použití takzvané Soli (Salt). [13] Sůl znamená, že se k heslu přidá ještě další sekvence znaků před tím, než je vytvořen hashový řetězec. V praxi se používají dva typy solí:

- Stejná sůl

  V rámci celé aplikace je použit jako sůl jeden řetězec. Ke každému heslu v databázi je tedy přidán stejný řetězec před vytvořením hashe.

- Specifická sůl pro každé heslo

V tomto případě je ke každému heslu při jeho vytvoření, i následném ověřování, přidán specifický řetězec jako sůl. Tento řetězec můžou tvořit například náhodně generované znaky.

5.4.5 Pomalé hashe

Poslední možností, která však čím dál více nabírá na popularitě, je použití pomalého hashování. Nejčastěji používané hashovací funkce, jako jsou SHA1 a MD5, jsou velice rychlé. Tento jev však není žádoucí, pokud je heslo pod takzvaným prolomením hrubou silou (Brute force attack). Používají se tedy hashovací metody, které jsou pomalé (například bcrypt) a tím pádem značně prodlužují dobu, která je potřeba k prolomení hesla.

5.4.6 Ochrana hesla v mé aplikaci

V mé aplikaci jsem se rozhodl ukládat hesla v zahashované podobě. Pro vyšší zabezpečení používám také sůl. Dále jsem se musel rozhodnout, jakou sůl použiji. Stejnou pro všechna hesla jsem použít nechtěl. Volba tedy padla na sůl generovanou samostatně pro každé heslo. Abych sůl neukládal v databázi a nevystavoval ji tak potencionálnímu útočníkovi, generuji ji z uživatelského jména. Při vytvoření hesla jsou první čtyři znaky z uživatelského jména přidány k heslu a celý tento řetězec je poté zahashován a uložen do databáze.

6 Testování aplikace

U každé aplikace, jež má být jakkoliv používána, je nutné provést testování, zda funguje bez chyb a zda splňuje svůj účel. Testovat aplikaci lze jak manuálně, tak pomocí automatických testů. V každém projektu by měly proběhnout oba dva způsoby testování, aby byla aplikace jak verifikována, tak validována. Verifikace zajišťuje, že je software dobře vyvinut z pohledu kvality kódu, tedy zda jsou ošetřeny vstupy, zda jsou poskytovány správné výstupy a obecně, zda je aplikace dostatečně robustní. Verifikace již však nezkontroluje, jestli je aplikace užitečná, tedy zda dělá skutečně to, k čemu je určena. [18] K tomuto účelu slouží validace, během níž by mělo dojít ke kontrole, že se software chová dle business požadavků.

6.1 Způsoby testování

Prvním způsobem je manuální testování, tedy hlavně testování v rámci uživatelského rozhraní, ale lze takto testovat například i REST služby. Manuální testování je vhodné pro validaci aplikace a také k takzvanému exploratory testingu. [19] Exploratory testing je metoda, kdy tester nepostupuje podle jasně daného scénáře, ale snaží se dojít k určitému výsledku nestandardními způsoby, hledá hranice funkcionality systému a obecně se snaží u testů navodit situace, které se nedají otestovat automaticky nebo neodpovídají standardnímu způsobu práce s aplikací.

Na druhé straně existuje také automatické testování. To je vhodné zejména k takzvaným unit testům. Unit testování probíhá na nejnižší úrovni, tedy na úrovni kódu. Tyto testy si píše sám vývojář a jejich tvorba by měla probíhat společně s vývojem kódu. Otestována by měla být každá veřejná metoda třídy. Privátní metody se netestují, protože to by porušilo princip objektově orientovaného programování zvaný zapouzdření. [21]

V případě aplikace vývojové metody zvané TDD, tedy Test Driven Development (Vývoj řízený testy) [20] probíhá vývoj tak, že programátor nejdříve napíše test pro funkcionalitu, která zatím neexistuje, a kterou se chystá vyvinout. První spuštění pochopitelně vygeneruje špatný výsledek. Poté je vyvinuta samotná funkcionalita a test je spuštěn znova. Pokud i tentokrát selže, musí se programátor k funkcionalitě vrátit a upravit ji. To musí opakovat do té doby, než test úspěšně projde. Test nadále zůstává v projektu, a proto bude další testování,v případě budoucí refaktorizace kódu, velice rychlé a spolehlivé.

Automatické testování lze provádět i v rámci grafického rozhraní. K tomuto účelu existují různé frameworky, jako například Selenium nebo Unified Functional Testing. Takové testy simulují chování uživatele, který s aplikací pracuje.

Automatické testy, ať již kódu, nebo uživatelského rozhraní jsou pak velmi užitečné v případě procesu zvaného průběžná integrace (Continous Integration). [22] Ta funguje na principu automatického sestavení aplikace a spuštění testů, kdykoliv některý z vývojářů na projektu pošle nový kód do systému správy verzí (například Git, Bitbucket). Pokud je průběh testů časově náročný, lze v nástroji, který má CI na starosti, nastavit, aby automatické sestavení a spuštění

testů probíhalo jednou denně, nejlépe v noci, kdy se nepředpokládá, že by měl přibýt nový kód, jenž nebude daný den otestován.

6.2 Testování v mém projektu

Při testování mé aplikace jsem postupoval několika způsoby. Na nejnižší úrovni mám napsány unit testy pro vybrané třídy. Dále jsem manuálně otestoval GUI a nakonec také rozhraní WebAPI.

6.2.1 Unit testy

Unit testy jsou takové testy, které mají otestovat nejmenší možnou ucelenou část kódu, která přijímá parametry, provádí funkcionalitu a vrací výsledek.[29] Tím, že tyto testy píše sám vývojář, najdou se základní chyby velice brzo a neovlivní tak ostatní části kódu. Tím se nejen zvyšuje rychlost vývoje, ale také se tím redukují náklady na vývoj, protože platí pravdilo, že čím dříve je chyba nalezena, tím je její oprava levnější. [29]

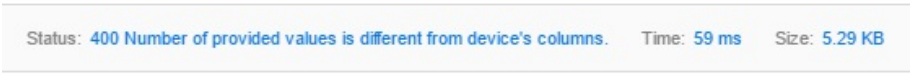
Z pohledu požadavků na správnou funkčnost aplikace je nejdůležitější business vrstva, v mém případě projekt ServiceLayer, testy jsem tedy napsal pro tuto vrstvu. Při psaní unit testů jsem využil několika nástrojů, které podporují rychlou, jednoduchou tvorbu takových testů. Tyto nástroje, NUnit, FakeItEasy a Fluent Assertions, budou popsány dále. Využité nástroje budou popsány dále.

6.2.2 Testování uživatelského rozhraní

Dalším testováním, které jsem provedl, bylo manuální testování v grafickém uživatelském rozhraní. V rámci tohoto testování jsem provedl několik scénářů:

- Jako uživatel s rolí User:
  - Přístup pouze k povoleným funkcím (tedy například nemožnost zobrazení administrace uživatelů)
  - Zobrazení pouze těch zařízení, ke kterým má uživatel oprávnění
  - Správné formátování zobrazených dat
  - Nemožnost přihlásit se jako uživatel s blokovaným nebo expirovaným účtem.
- Jako uživatel s rolí Admin
  - Zobrazení všech stránek v projektu
  - Zobrazení/vytváření/editace/mazání zařízení
  - Zobrazení/vytváření/editace/mazání uživatelů
  - Zobrazení dat pro všechna zařízení, tedy nevyužití oprávnění

Obrázek 8: Zobrazení odpovědi o chybějících parametrech v požadavku z API v aplikaci Postman



Obrázek 9: Zobrazení odpovědi o různém počtu hodnot z API v aplikaci Postman

- Testy bez návaznosti na uživatelskou roli
  - Zadání validních hodnot ve všech formulářích
  - Kontrola nemožnosti vložení nevalidních dat ve všech formulářích (ošetření vstupů)
  - Správné formátování zobrazených stránek
  - Kontrola nadpisů a ostatních textů na překlepy

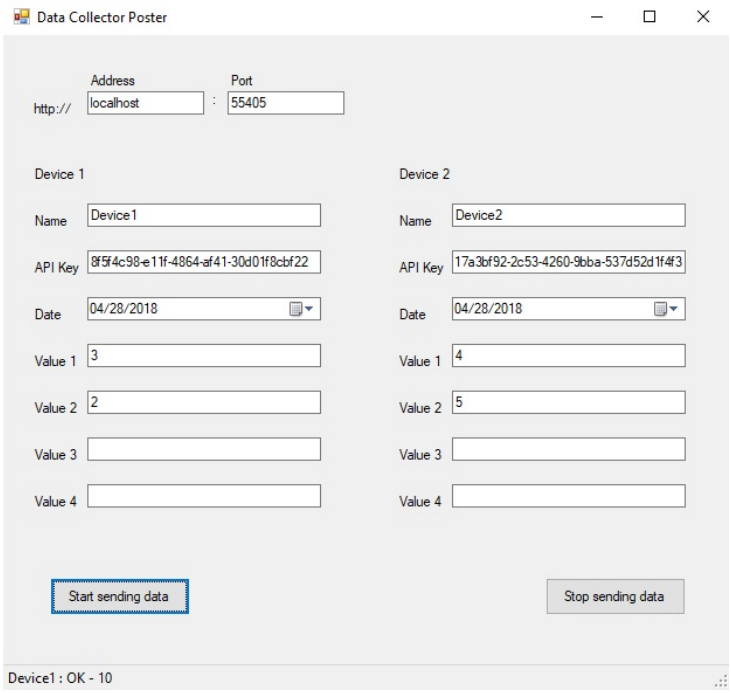
6.2.3 Testování WebAPI

Poslední testy, které jsem provedl, byly testy odesílání dat přes WebAPI. K tomuto jsem využil volně dostupný nástroj Postman. Tato aplikace poskytuje celou řadu nástrojů pro kompletní testování API rozhraní. V mém projektu jsem využil komponentu Postman Collections, která umožňuje vytváření webových požadavků, včetně vložení parametrů, a zobrazení odpovědi získané a API testované aplikace.

Testované scénáře byly:

- Požadavek se všemi platnými parametry - odpověď OK
- Požadavek s chybějícím parametrem apiKey nebo deviceName - odpověď o špatném požadavku (obrázek 8)
- Požadavek s různým počtem hodnot, než je uvedeno sloupců u zařízení (obrázek 9)
- Požadavek s vyplněným datem - v databázi uloženo datum z požadavku
- Požadavek bez vyplněného data - v databázi uloženo datum přijetí požadavku

Také jsem potřeboval otestovat paralelní příjem z více zařízení najednou. Vytvořil jsem si tedy velmi jednoduchou aplikaci, do které se zadají informace pro dvě zařízení a ta pak v samostatných vláknech odesílá data do mé hlavní aplikace. GUI této aplikace na obrázku 10. Tento test mi nejdříve končil s takovým výsledkem, že aplikace spadla z důvodu paralelního zápisu do databáze. Problém jsem vyřešil, jak bylo popsáno v části 2.9.1.



Obrázek 10: Prostředí aplikace pro test paralelního zápisu z dvou zařízení

6.2.4 Pokrytí kódu testy

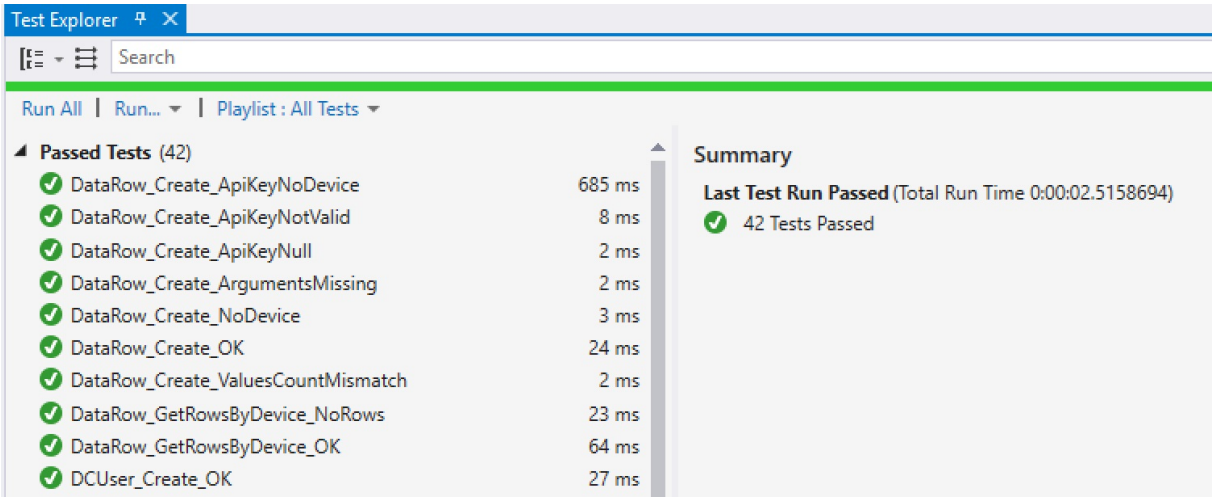
Jednou z metrik, které nabízi Visual Studio, je pokrytí kódu testy (Test Coverage). Jedná se procentuální hodnotu, kolik napsaného kódu je pokryto testy, které jsou vytvořeny. Soustředit se na číslo samotné, aby bylo co nejvyšší, není zcela správné. [24] Místo toho by se vývojář měl soustředit na kvalitu testů a metriku Test Coverage brát spíše jako informační hodnotu. Obecně by se dalo říci, že hodnota pod 50% značí problém a indikuje nedostatečné pokrytí. Výsledná hodnota by měla být kolem 80-90%, ovšem jak jsem již zmínil, primární cíl by neměl být dostat se na požadovanou hodnotu, ale spíše se soustředit na kvalitu testů samotných.

Jak jsem již zmínil v kapitole 6.2.1, soustředil jsem se na testy pokrývající business vrstvu aplikace. Se zaměřením na obsah testů samotných a ne jejich počet, jsem dle Visual Studia nakonec pokryl 80,60% kódu.

6.3 NUnit

NUnit je framework pro tvorbu, vyhledání a spouštění unit testů pro všechny jazyky platformy .NET. [25]. Testy lze spouštět přes konzoli, případně, po instalaci rozšíření Test Adapter, také přímo ve Visual Studiu.

NUnit nevyžaduje žádný oddělený projekt v rámci řešení, jako to dělá například MsTest od společnosti Microsoft, nicméně ve svém řešení jsem takový projekt vytvořil, s názvem TestApp. Framework pak automaticky vyhledává všechny testy v projektu. Při psaní testů je potřeba



Obrázek 11: Ukázka výpisu výsledků testů spuštěných pomocí NUnit frameworku

použít Attributy, dle kterých NUnit zjistí, že se jedná o třídu s testy testy samotné. Nejdůležitější použitelné Attributy, které jsem zároveň použil v projektu, jsou:

- `[TestFixture]` - označuje třídu, ve které se nacházejí testy
- `[Setup]` - metoda, která je volána před každým testem. Může obsahovat například inicializaci proměnných.
- `[Test]` - označení pro metodu, že se jedná o spustitelný test

V mém řešení jsem využil rozšíření Test Adapter a testy spouštím přímo v prostředí Visual Studia. NUnit sám vyhledá testy a vypíše jejich seznam společně s výsledky a informací o chybě, pokud některý z testů neskončí úspěšně.

6.4 FakeItEasy

Důležitým aspektem při tvorbě testů je soustředit se na funkčnost konkrétní metody, pro kterou se test píše. Taková metoda však může v rámci svého kódu volat další třídy, například pro zápis dat do databáze. Ten však již nespadá do rozsahu testu samotného. V takových případech se používá návrhový vzor Mock Object. [26]

Vytvoření Mock objektu znamená, že vytvoříme atrapu objektu, kterou pak podsuneme jako závislost do naší testované třídy. Vytvořené atrapě můžeme jednoduše nastavit očekávaný výsledek, který má vrátit do testované třídy, bez jakékoliv další funkcionality, jelikož ta není předmětem testu.

Tímto přístupem získáme několik výhod, mezi ty hlavní patří:

- Rychlejší psaní testů
- Zredukujeme, případně zcela zrušíme závislosti

- Testy poběží rychleji, protože se vyhneme spouštění reálných metod, které mohou být časově náročnější (například zmíněný zápis do databáze)

Mock objekty si můžeme vytvářet sami, můžeme však také využít automatizovaných nástrojů, které celý proces zjednodušují, a tedy zrychlují. Ve své práci jsem použil využil takového nástroje, který se jmenuje FakeItEasy. S pomocí tohoto nástroje je vytvoření atrapy objektu otázkou jednoho řádku kódu. Atrapa je vytvořena a může být použita, avšak neobsahuje v sobě žádnou funkcionalitu. Tu můžeme opět velmi jednoduše nastavit. Ukázka vytvoření atrapy dvou objektů a nastavení jednoho z nich jako návratové hodnoty jedné z metod druhého je uvedena ve výpisu kódu číslo 5

```
private IDataRowRepository dataRowRepository = A.Fake<IDataRowRepository>();
private IEnumerable<DataRowFull> device1Rows = device1Rows = A.Fake<IEnumerable
    <DataRowFull>>();

A.CallTo(() => dataRowRepository.GetDataRowsByDevice("device1")).Returns(
    device1Rows);
```

Výpis 5: Ukázka vytvoření Mock Objectu a nastavení návratové hodnoty pro jednu jeho metodu

6.5 Fluent Assertions

K vyhodnocení, zda test skončí úspěchem nebo neúspěchem, poskytuje framework NUnit mechanismus zvaný Assert, tedy očekávání. Ten funguje tak, že po zavolání testované metody s předem stanovenými parametry je očekáván (asertován) konkrétní výstup. Krom NUnit existují také další nástroje, které jsou specializované konkrétně pouze na asertace. Jedním z takových nástrojů, které jsem použil ve své práci, je program Fluent Assertions.

Nástroj Fluent Assertions je soubor rozšiřujících metod (extension methods), které umožňují přirozeněji definovat očekávané výsledky v unit testech. [27] V samotném testu se nepoužívá konstrukce `Assert`, ale testované třídy jsou rozšířeny o metody, které tyto asertace poskytují. Očekávané výstupy je také možno řetězit. Základní pravidlo psaní unit testů je, že v každém testu by měla existovat pouze jedna assertace. Řetězení assertací v rámci Fluent Assrtions je výhodné ve chvíli, kdy očekáváme, že metoda vyhodí výjimku. V takovém případě očekáváme nejen vyhození výjimky, ale také konkrétní zprávu v dané výjimce. Ukázka takové asertace je ve výpisu kódu číslo 6.

```
create.Should().Throw<ArgumentException>().And.Message.Should().Contain(
    ServiceLayer.Resources.Exceptions.ArgumentsMissing);
```

Výpis 6: Assertace vyhození výjimky včetně textu zprávy

6.6 Zhodnocení testování

Ve své práci jsem testování věnoval podstatnou část celého vývoje. Výsledkem je aplikace, která je jak verifikovaná, tak validovaná a její vrstva starající se o business logiku, ServiceLayer, má připravenou sadu celkem 43 testů, které objevila chyby již během vývoje, ale také je připravena pro případ, že bude aplikace v budoucnu refaktorována.

7 Závěr

Cílem mé práce bylo bylo vytvořit aplikaci, která bude schopná přijímat data od různých zařízení, ukládat je do databáze a zobrazovat je na vyžádání uživatelům. Výsledná aplikace tyto požadavky splňuje. Zároveň jsem se v práci zaměřil na použité metody a nástroje, abych mohl zhodnotit vhodnost zvoleného postupu.

Vybraná architektura aplikace byla na implementaci náročnější a bylo možné zvolit jednodušší, rychlejší postup. Takový přístup by však byl výhodný pouze pro jednorázový vývoj. V mém případě jsem zvolil prvotní cestu sice složitější, avšak s nespornými výhodami v případě, že by v budoucnu bylo třeba aplikaci upravovat.

Zvolený formát ukládání dat se ukázal jako velice efektivní, ať již při práci s daty, nebo v případě změn v databázi samotné. Taktéž zvolený podpůrný nástroj se ukázal velmi přínosný a při vývoji ušetřil spoustu práce.

Testování proběhlo tak, jak by mělo v praxi probíhat. První testy byly psány již v brzké fázi vývoje, nalezené chyby byly tedy odstraněny velice brzy. Testování pak provázelo celý vývoj až do konce a chyby byly opravovány průběžně. Vybrané nástroje hodnotím také velice kladně, vzhledem k jejich přínosu pro urychlení a zkvalitnění celého vývoje.

V průběhem vývoje a jeho výsledkem jsem tedy velmi spokojen a jsem přesvědčen, že se osvědčil jako správná cesta při vývoji aplikace daného typu a rozsahu.

Literatura

[1] Software Architecture Patterns, Mark Richards, 2015 [online] Dostupné z www <http://www.oreilly.com/programming/free/software-architecture-patterns.csp>

[2] DevIq [online] 2018 [cit. 22.4.2018] Repository pattern. Dostupné z WWW: <http://deviq.com/repository-pattern/>

[3] DevIq [online] 2018 [cit. 22.4.2018] YAGNI. Dostupné z WWW <http://deviq.com/yagni/>

[4] Patterns of Enterprise Application Architecture, Martin Fowler [online] 2002 [cit. 22.4.2018] Dostupné z www <https://martinfowler.com/eaaCatalog/index.html>

[5] NAGEL, Christian. "C# 2005–programujeme profesionálně.[sl]."(2007).

[6] codetuple [online] [cit. 25.4.2018] IoC containers, a comparison. Dostupné z WWW <http://www.codetuple.com/articles/aspnet/HQMvY32Nzxp/ioc-containers-a-comparison-1/>

[7] TutorialsTeacher [online] 2018 [cit. 25.4.2018] What is Web API? Dostupné z WWW <http://www.tutorialsteacher.com/webapi/what-is-web-api>

[8] REST API Tutorial [online] [cit. 25.4.2018] REST Architectural Constraints. Dostupné z WWW <https://restfulapi.net/rest-architectural-constraints/>

[9] DB-Engines [online] 2018 [cit. 25.4.2018] DB-Engines Ranking. Dostupné z WWW <https://db-engines.com/en/ranking>

[10] Microsoft Developer Network [online] 2016 [cit. 26.4.2018] Entity Framework Version History. Dostupné z WWW <https://msdn.microsoft.com/en-us/library/jj574253(v=vs.113).aspx>

[11] Nullable Code [online] 2013 [cit. 26.4.2018] Splitting Entity Framework Model classes into separate projects. Dostupné z WWW <http://nullablecode.com/2013/09/splitting-entity-framework-model-classes-separate-projects/>

[12] Internet Engineering Task Force [online] 2005 [cit. 26.4.2108] RFC 4122. Dostupné z WWW <https://www.ietf.org/rfc/rfc4122.txt>

[13] lifehacker [online] 2012 [cit. 27.4.2018] How Your Passwords Are Stored on the Internet (and When Your Password Strength Doesn't matter). Dostupné z WWW <https://lifehacker.com/5919918/how-your-passwords-are-stored-on-the-internet-and-when-your-password-strength-doesnt-matter>

[14] Lifewire [online] 2018 [cit. 28.4.2018] Rainbow Tables: Your Password's Worst Nightmare. Dostupné z WWW <https://www.lifewire.com/rainbow-tables-your-passwords-worst-nightmare-2487288>

[15] Symantec [online] [cit. 28.4.2018] What is SSL, TLS and HTTPS? Dostupné z WWW <https://www.websecurity.symantec.com/security-topics/what-is-ssl-tls-https>

[16] SSLS [online] [cit. 28.4.2018] TLS. Dostupné z WWW <https://www.ssls.cz/slovnik/tls.html>

[17] SSLS [online] [cit. 28.4.2018] SNI. Dostupné z WWW <https://www.ssls.cz/slovnik/sni.html>

[18] Tools QA [online] 2017 [cit. 28.4.2018] Difference between Verification and Validation. Dostupné z WWW <http://toolsqa.com/software-testing/difference-between-verification-and-validation/>

[19] ISTQB Exam Certification [online] [cit. 28.4.2018] What is xploratory testing in software testing? Dostupné z WWW <https://www.guru99.com/exploratory-testing.html>

[20] Agile Data [online] [cit. 28.4.2018] Introduction to Test Driven Development (TDD). Dostupné z WWW <http://agiledata.org/essays/tdd.html>

[21] CodeAhoy [online] 2016 Should You Unit Test Private Methodds? Dostupné z WWW <https://codeahoy.com/2016/11/19/should-you-unit-test-private-methods/>

[22] Visual Studio [online] [cit. 28.4.2018] What is Continous Integration? Dostupné z WWW <https://www.visualstudio.com/learn/what-is-continuous-integration/>

[23] Postman [online] [cit. 28.4.2018] Postman Collections. Dostupné z WWW <https://www.getpostman.com/collection>

[24] Martin Fowler [online] 2012 [cit. 28.4.2018] TestCoverage. Dostupné z WWW <https://martinfowler.com/bliki/TestCoverage.html>

[25] NUnit [online] [cit. 28.4.2018] Dostupné z WWW <http://nunit.org/>

[26] Net Objectives [online] 2015 [cit. 29.4.2018] TheMockObjectPattern. Dostupné z WWW <http://www.netobjectives.com/PatternRepository/index.php?title=TheMockObjectPattern>

[27] Fluent Assertions [online] [cit. 29.4.2018] Dostupné z WWW <https://fluentassertions.com/>

[28] Michal Franc [online] 2013 [cit. 29.4.2018] Good unit test - One Assert. Dostupné z WWW <https://mfranc.com/unit-testing/good-unit-test-one-assert/>

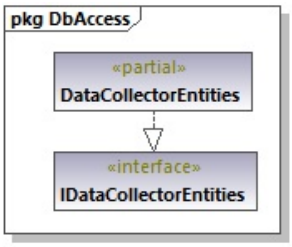
[29] Software Testing Fundamentals [online] [cit. 29.4.2018] Unit Testing. Dostupné z WWW <http://softwaretestingfundamentals.com/unit-testing/>

A Výpis unit testů

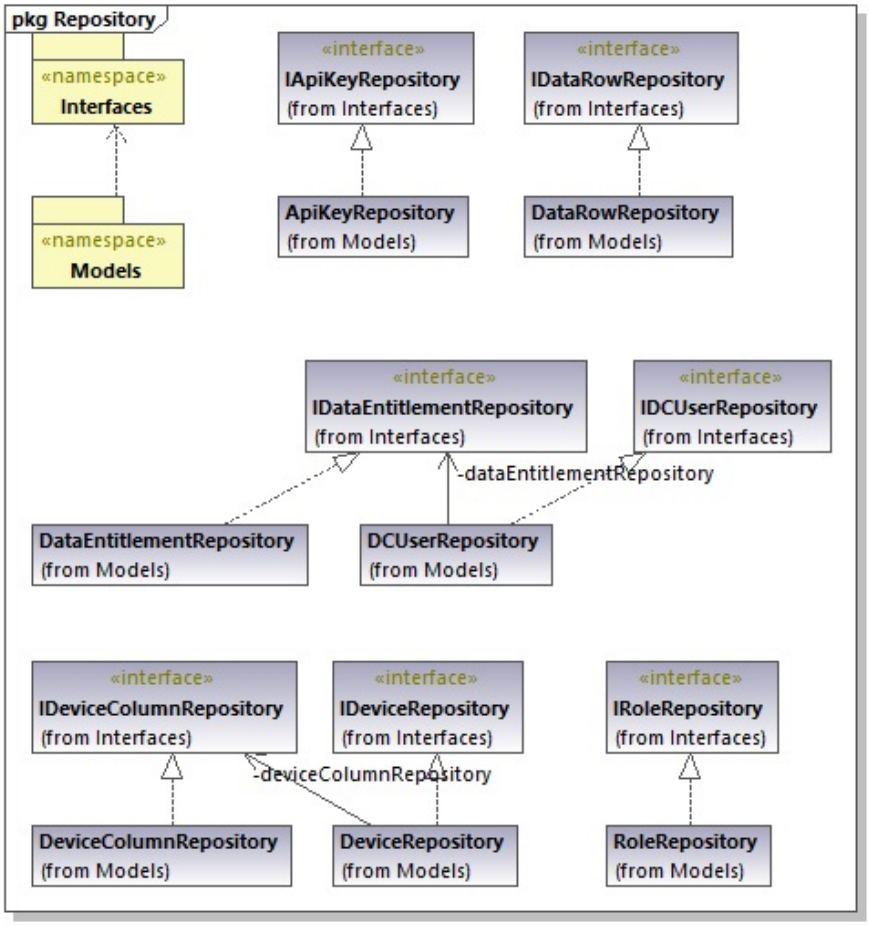
- DataServiceTests
  - DataRow_GetRowsByDevice_OK
  - DataRow_GetRowsByDeviceAndDate_OK
  - DataRow_GetRowsByDevice_NoRows
  - DataRow_Create_OK
  - DataRow_Create_ArgumentsMissing
  - DataRow_Create_ApiKeyNull
  - DataRow_Create_ApiKeyNotValid
  - DataRow_Create_ApiKeyNoDevice
  - DataRow_Create_NoDevice
  - DataRow_Create_ValuesCountMismatch
- DCUserServiceTests
  - DCUser_GetDCUsers
  - DCUser_GetByUsername_Found
  - DCUser_GetByUsername_NotFound
  - DCUser_DeleteDCUser
  - DCUser_IsUserValid_OK
  - DCUser_IsUserValid_Expired
  - DCUser_IsUserValid_Blocked
  - DCUser_Create_ShortUsername
  - DCUser_Create_OK
  - DCUser_Update_OK
  - DCUser_GetByUsernameAndPass_Found
- DeviceServiceTests
  - Device_GetDevices
  - Device_GetDevicesForUser
  - Device_GetDeviceByName
  - Device_DeleteDevice
  - Device_IsDeviceValid_True

  - Device_IsDeviceValid_False
  - Device_CreateDevice_OK
  - Device_CreateDevice_WrongName
  - Device_CreateDevice_WrongColumnCount
  - Device_CreateDevice_ColumnOrdersNotUnique
  - Device_CreateDevice_ColumnOrdersNotNumbers
  - Device_CreateDevice_ColumnNamesNotUnique
  - Device_UpdateDevice
- LoginServiceTests
  - ValidateUser_OK
  - ValidateUser_Blocked
  - ValidateUser_Expired
- PassHashingServiceTests
  - PassHashing_GetHash
  - PassHashing_Validate_True
  - PassHashing_Validate_False
- RoleServiceTests
  - GetRole_ReturnNull
  - GetRole_ReturnRole
- UtilServiceTests
  - GetColumnTypes

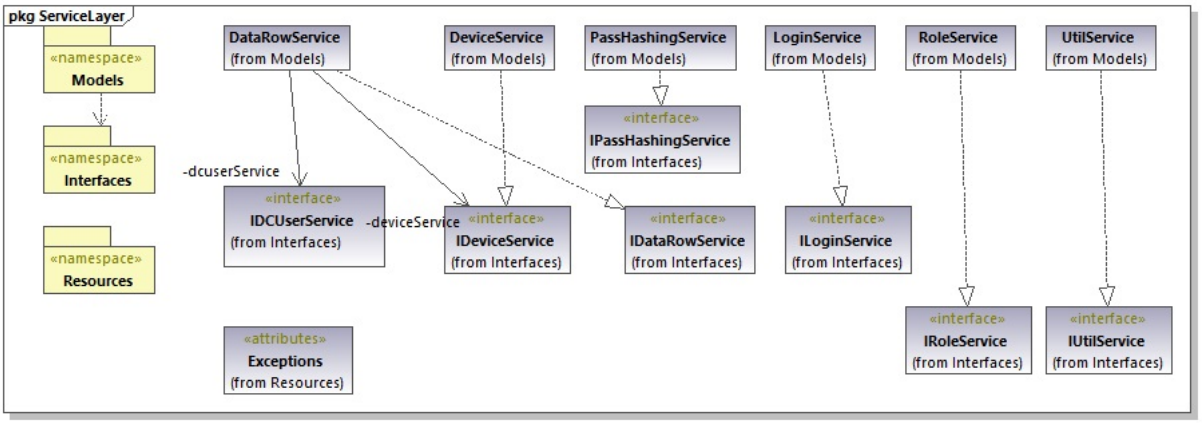
B Třídní diagramy



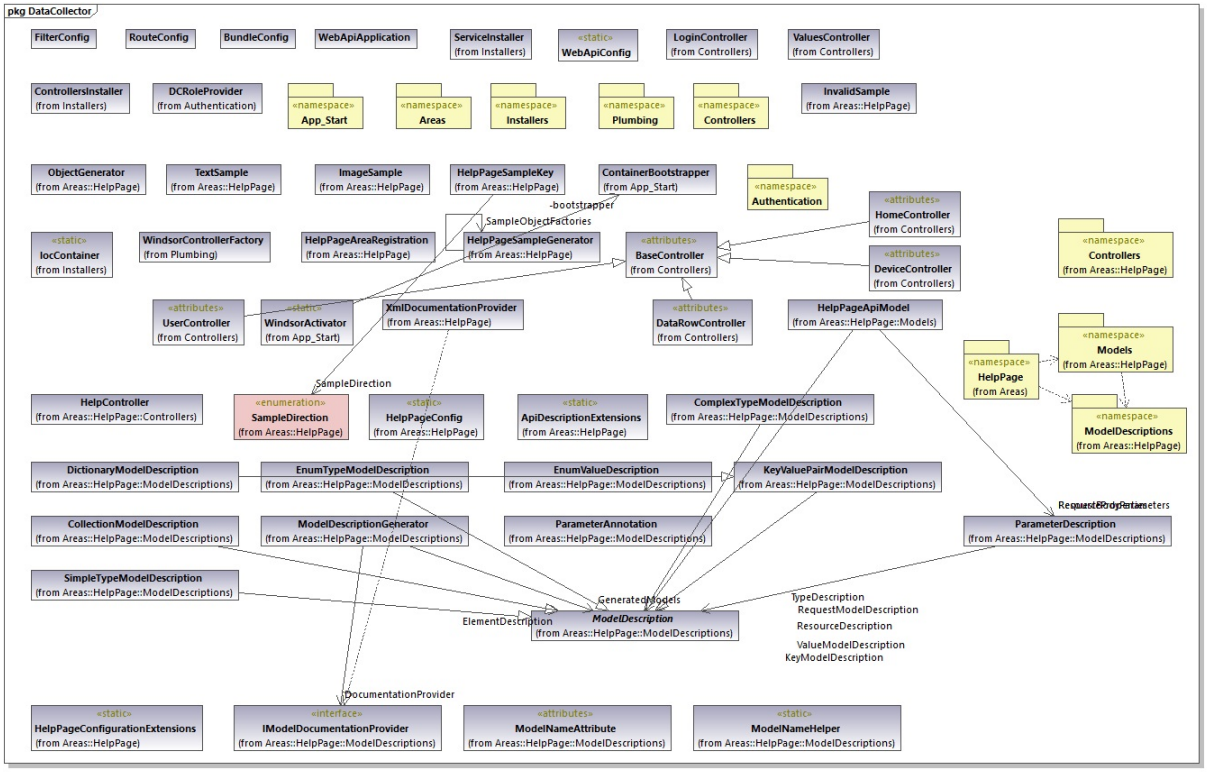
Obrázek 12: Třídní diagram pro vrstvu DBAccess



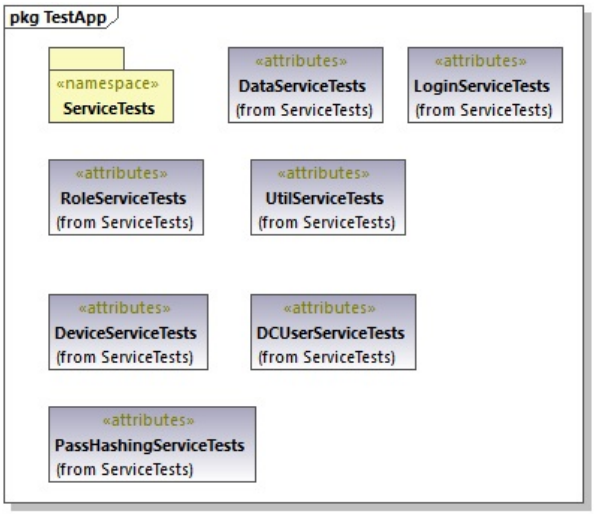
Obrázek 13: Třídní diagram pro vrstvu Repository



Obrázek 14: Třídní diagram pro vrstvu ServiceLayer



Obrázek 15: Třídní diagram pro vrstvu Web



Obrázek 16: Třídní diagram pro testovací projekt TestApp